Sử dụng hàm t.substr(i, s.size()) để tạo xâu con và so sánh trực tiếp với xâu mẫu s trong chuỗi văn bản t là một cách giải quyết hợp lý, và cách này có thể hoạt động bình thường. Tuy nhiên, điều quan trọng cần lưu ý là nếu chỉ sử dụng phương pháp này, độ phức tạp thời gian của bạn sẽ vẫn là **O(n \* m)**, do phải tạo và so sánh từng xâu con ở mỗi bước.

**Phương pháp sử dụng substr():**

Chúng ta sẽ sử dụng hàm substr(i, s.size()) để lấy xâu con bắt đầu tại vị trí i và có độ dài bằng với độ dài của chuỗi mẫu s. Sau đó, ta so sánh xâu con này với xâu mẫu s. Nếu hai xâu bằng nhau, ta tăng biến count.

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main() {

// Đọc dữ liệu từ file

ifstream input("cstring.inp");

ofstream output("cstring.out");

string s, t;

input >> s >> t;

int n = t.length();

int m = s.length();

int count = 0;

// Duyệt qua tất cả các vị trí trong t và lấy xâu con có độ dài m

for (int i = 0; i <= n - m; i++) {

// Lấy xâu con từ t bắt đầu tại vị trí i có độ dài m

if (t.substr(i, m) == s) {

count++;

}

}

// Ghi kết quả ra file

output << count << endl;

return 0;

}

**Giải thích:**

* **t.substr(i, m)**: Hàm này sẽ lấy xâu con từ chuỗi t bắt đầu từ vị trí i với độ dài m (độ dài của chuỗi mẫu s).
* **So sánh**: Sau khi lấy xâu con, ta so sánh nó với chuỗi mẫu s. Nếu chúng bằng nhau, nghĩa là chúng ta đã tìm thấy một lần xuất hiện của s trong t.
* **Biến đếm (count)**: Mỗi lần tìm thấy sự khớp, ta tăng biến count.

**Độ phức tạp:**

* **Hàm substr(i, m)**: Lấy xâu con từ vị trí i đến i + m tốn thời gian O(m), vì phải sao chép một đoạn con dài m từ chuỗi t.
* **Vòng lặp**: Vòng lặp duyệt qua tất cả các vị trí trong chuỗi t từ 0 đến n - m, tức là có tối đa n - m + 1 bước.

Do đó, tổng độ phức tạp thời gian của phương pháp này là **O(n \* m)**, trong đó:

* **n** là độ dài của chuỗi văn bản t.
* **m** là độ dài của chuỗi mẫu s.

**Đánh giá:**

* Phương pháp này rất dễ hiểu và thực hiện được, nhưng không tối ưu khi phải xử lý các chuỗi có kích thước lớn (đến 10610^6106).
* Việc tạo xâu con bằng hàm substr và so sánh sẽ làm tăng độ phức tạp và có thể rất tốn kém về mặt thời gian khi n và m đều rất lớn.

**So sánh với KMP:**

* Thuật toán **KMP** có độ phức tạp là **O(n + m)**, vì vậy nó sẽ tối ưu hơn khi xử lý với chuỗi có độ dài lớn. KMP sử dụng bảng LPS để tránh việc so sánh lại các phần đã khớp trước đó, điều này giúp tiết kiệm thời gian so với việc sử dụng substr trong phương pháp brute-force.

**Lý do phương pháp KMP tối ưu hơn:**

1. **Tránh sao chép và so sánh lại**: Trong KMP, chúng ta không cần phải sao chép xâu con mỗi lần mà chỉ so sánh trực tiếp từng ký tự. Điều này giúp giảm chi phí xử lý.
2. **Sử dụng bảng LPS**: Bảng LPS cho phép "nhảy" qua những vị trí không cần phải so sánh lại khi gặp sự không khớp, thay vì phải quay lại từ đầu mỗi lần.
3. **Độ phức tạp thấp hơn**: KMP có độ phức tạp O(n + m), trong khi phương pháp sử dụng substr có độ phức tạp O(n \* m).

**Kết luận:**

Phương pháp dùng substr kết hợp với vòng lặp là cách dễ hiểu và dễ triển khai, nhưng không phải là giải pháp tối ưu cho bài toán tìm kiếm chuỗi con, đặc biệt khi dữ liệu có kích thước rất lớn. Nếu bạn muốn tối ưu hóa chương trình và xử lý được các chuỗi lớn, **thuật toán KMP** là lựa chọn tốt hơn với độ phức tạp O(n + m).

// Hàm tính bảng LPS (Longest Prefix Suffix) cho mẫu S

vector<int> computeLPS(const string &pattern) {

int m = pattern.length(); // Lấy độ dài của mẫu

vector<int> lps(m, 0); // Mảng lps chứa các giá trị của bảng LPS, khởi tạo bằng 0

int len = 0; // Biến 'len' giữ độ dài tiền tố dài nhất đến thời điểm hiện tại

int i = 1; // Bắt đầu từ vị trí thứ 1 trong chuỗi mẫu (vì lps[0] luôn là 0)

// Tính bảng LPS cho mẫu

while (i < m) { // Tiến hành duyệt qua chuỗi mẫu

if (pattern[i] == pattern[len]) { // Nếu ký tự hiện tại trong mẫu bằng ký tự tiền tố dài nhất

len++; // Tăng độ dài tiền tố

lps[i] = len; // Gán giá trị của LPS tại vị trí i

i++; // Tiến sang ký tự tiếp theo

} else { // Nếu không khớp

if (len != 0) { // Nếu len khác 0, thì không cần phải di chuyển i mà quay lại vị trí của tiền tố

len = lps[len - 1]; // Quay lại vị trí của tiền tố dài nhất trước đó

} else { // Nếu len == 0, nghĩa là không có tiền tố nào phù hợp

lps[i] = 0; // Gán lps[i] là 0

i++; // Tiến sang ký tự tiếp theo

}

}

}

return lps; // Trả về bảng LPS

}

// Hàm tìm kiếm xâu S trong xâu T bằng thuật toán KMP

int KMPSearch(const string &text, const string &pattern) {

int n = text.length(); // Độ dài chuỗi văn bản

int m = pattern.length(); // Độ dài chuỗi mẫu

if (m == 0 || n == 0) return 0; // Nếu một trong hai chuỗi rỗng, trả về 0

// Tính bảng LPS cho chuỗi mẫu

vector<int> lps = computeLPS(pattern);

int i = 0, j = 0; // i là chỉ số trong văn bản (text), j là chỉ số trong mẫu (pattern)

int count = 0; // Biến đếm số lần xuất hiện của mẫu trong văn bản

// Duyệt qua tất cả các ký tự trong văn bản

while (i < n) {

if (pattern[j] == text[i]) { // Nếu ký tự trong mẫu bằng ký tự trong văn bản

i++; // Tiến đến ký tự tiếp theo trong văn bản

j++; // Tiến đến ký tự tiếp theo trong mẫu

}

// Nếu đã khớp toàn bộ mẫu

if (j == m) {

count++; // Tăng biến đếm vì đã tìm thấy một lần khớp

j = lps[j - 1]; // Sử dụng bảng LPS để tiếp tục tìm kiếm từ vị trí tiếp theo trong mẫu

}

// Nếu không khớp

else if (i < n && pattern[j] != text[i]) {

if (j != 0) { // Nếu j không phải là 0, thì dùng bảng LPS để "nhảy" qua vị trí đã khớp

j = lps[j - 1]; // Di chuyển đến vị trí của tiền tố dài nhất trước đó

} else { // Nếu j == 0, chỉ cần tiếp tục duyệt qua văn bản

i++; // Tiến sang ký tự tiếp theo trong văn bản

}

}

}

return count; // Trả về số lần tìm thấy mẫu trong văn bản

}

int main() {

// Đọc dữ liệu từ file

ifstream input("cstring.inp"); // Mở file input để đọc dữ liệu

ofstream output("cstring.out"); // Mở file output để ghi kết quả

string S, T; // Khai báo hai chuỗi: S (mẫu) và T (văn bản)

input >> S >> T; // Đọc chuỗi S và T từ file

// Tính số lần xuất hiện của S trong T

int result = KMPSearch(T, S);

// Ghi kết quả vào file output

output << result << endl;

return 0; // Kết thúc chương trình

}

**1. Định nghĩa bảng LPS:**

Bảng **LPS** (Longest Prefix Suffix) chứa các giá trị chỉ ra độ dài của tiền tố dài nhất (prefix) đồng thời là hậu tố (suffix) của chuỗi mẫu cho mỗi vị trí trong chuỗi. Mỗi giá trị tại vị trí i trong bảng LPS sẽ cho biết độ dài của phần tiền tố dài nhất của chuỗi mẫu S[0...i] cũng là hậu tố của chuỗi đó.

**2. Cấu trúc của bảng LPS:**

Giả sử chuỗi mẫu S = "ababa". Chúng ta sẽ xây dựng bảng LPS như sau:

* Bảng LPS sẽ có kích thước bằng độ dài của chuỗi mẫu S, tức là có 5 phần tử.
* Mỗi phần tử lps[i] sẽ lưu trữ độ dài của tiền tố dài nhất của chuỗi S[0..i] mà cũng là hậu tố của chuỗi này.

**3. Bước 1: Khởi tạo:**

* Chúng ta sẽ khởi tạo một mảng lps với tất cả giá trị bằng 0: lps = [0, 0, 0, 0, 0].
* len sẽ lưu độ dài của tiền tố dài nhất đã gặp cho đến thời điểm hiện tại. Ban đầu, len = 0.
* i là chỉ số duyệt qua chuỗi mẫu, ban đầu i = 1 (vì lps[0] luôn là 0).

**4. Bước 2: Tính bảng LPS:**

Dưới đây là quá trình tính toán bảng LPS cho chuỗi mẫu S = "ababa":

**i = 1:**

* So sánh S[i] = "b" với S[len] = "a". Chúng không khớp.
* Vì len == 0, ta đặt lps[i] = 0 và tăng i: lps = [0, 0, 0, 0, 0].

**i = 2:**

* So sánh S[i] = "a" với S[len] = "a". Chúng khớp.
* Ta tăng len lên 1 và gán lps[i] = len, tức là lps[2] = 1.
* Tăng i: lps = [0, 0, 1, 0, 0].

**i = 3:**

* So sánh S[i] = "b" với S[len] = "b". Chúng khớp.
* Ta tăng len lên 2 và gán lps[i] = len, tức là lps[3] = 2.
* Tăng i: lps = [0, 0, 1, 2, 0].

**i = 4:**

* So sánh S[i] = "a" với S[len] = "a". Chúng khớp.
* Ta tăng len lên 3 và gán lps[i] = len, tức là lps[4] = 3.
* Tăng i: lps = [0, 0, 1, 2, 3].

**5. Kết quả bảng LPS:**

Sau khi duyệt hết chuỗi, bảng LPS cuối cùng sẽ là:

css

Sao chép mã

lps = [0, 0, 1, 2, 3]

**6. Giải thích bảng LPS:**

Bảng LPS [0, 0, 1, 2, 3] có ý nghĩa như sau:

* **lps[0] = 0**: Với chuỗi con S[0] = "a", không có tiền tố nào là hậu tố.
* **lps[1] = 0**: Với chuỗi con S[0..1] = "ab", không có tiền tố nào là hậu tố.
* **lps[2] = 1**: Với chuỗi con S[0..2] = "aba", tiền tố "a" là hậu tố.
* **lps[3] = 2**: Với chuỗi con S[0..3] = "abab", tiền tố "ab" là hậu tố.
* **lps[4] = 3**: Với chuỗi con S[0..4] = "ababa", tiền tố "aba" là hậu tố.

**7. Cách sử dụng bảng LPS trong thuật toán KMP:**

Bảng LPS giúp thuật toán KMP không phải quay lại đầu mỗi lần không khớp, mà "nhảy" đến một vị trí mà ta biết chắc rằng một phần của mẫu đã khớp. Khi gặp sự không khớp, thay vì quay lại vị trí đầu tiên của mẫu, ta dùng bảng LPS để xác định tiếp tục so sánh từ vị trí nào trong mẫu.

Ví dụ:

* Khi tìm kiếm mẫu "ababa" trong một chuỗi văn bản, nếu ta gặp một sự không khớp sau khi đã khớp được "ab", thay vì bắt đầu lại từ đầu mẫu, ta có thể dùng bảng LPS để biết rằng ta có thể tiếp tục so sánh từ vị trí thứ 2 trong mẫu (vì "ab" là tiền tố và hậu tố của mẫu).

**8. Tóm lại:**

Bảng LPS là một công cụ mạnh mẽ giúp thuật toán KMP tìm kiếm chuỗi mẫu một cách hiệu quả. Với chuỗi mẫu S = "ababa", bảng LPS của nó sẽ là [0, 0, 1, 2, 3], và mỗi giá trị trong bảng này chỉ ra độ dài tiền tố dài nhất mà cũng là hậu tố của chuỗi con S[0..i]. Bảng này giúp thuật toán KMP tránh phải so sánh lại những phần đã khớp trước đó, làm tăng hiệu suất của quá trình tìm kiếm.